Algorithms and Data Structures 2
CS 1501

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Midterm grades posted (out of 60)
  - Question reattempts to get up to **7 points** back
  - Please use GradeScope's Regrade Requests for each question individually **due on Monday 4/17 at 11:59 pm**
- Upcoming Deadlines
  - Lab 10: Tuesday 4/11 @ 11:59 pm
  - Homework 11: **next Friday** @ 11:59 pm
  - Assignment 4: Friday 4/14 @ 11:59 pm
    - Support video and slides on Canvas + Solutions for Labs 8 and 9

Previous lecture

- Minimum Spanning Tree (MST)
  - Prim’s MST algorithm
    - naiive implementation
    - Best Edges array implementation
    - using a min-heap
  - Kruskal’s MST algorithm

This Lecture

- Weighted Shortest Paths problem
  - Dijkstra’s single-source shortest paths algorithm
  - Bellman-Ford’s shortest paths algorithm

Kruskal's MST algorithm

- Insert all *e* edges into a PQ
- T = an empty set of edges
- Repeat until T contains **v-1** edges
  - Remove a min edge from the PQ
  - Add the edge to T if the edge does not create a cycle in T
- return T

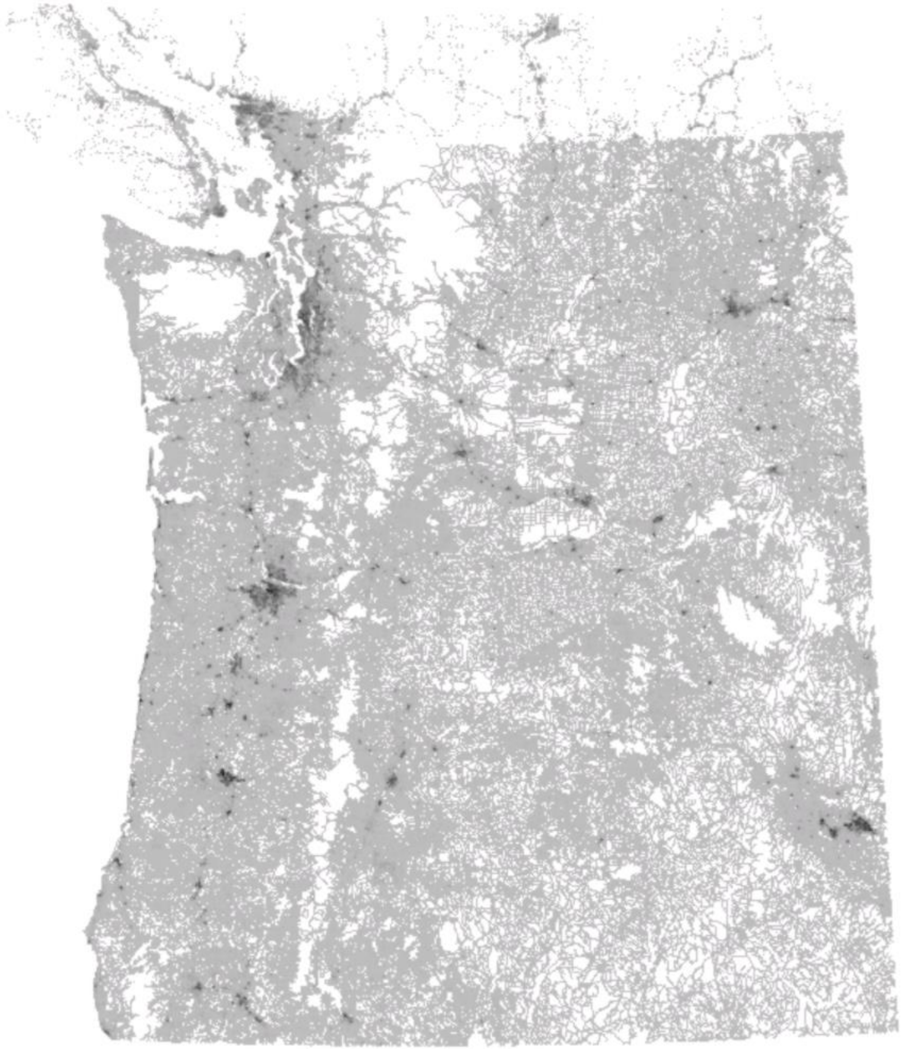
Runtime of Kruskal's MST algorithm

- Instead of building up the MST starting from a single vertex, we build it up using edges all over the graph
- How do we efficiently implement cycle detection?
  - BFS/DFS
    - v + e
  - Union/Find data structure (not covered)
    - log v

Kruskal's Runtime

- e iterations
  - removeMin → log e
  - Cycle detection
    - v + e using DFS/BFS
    - log v using Union/Find
- Total runtime = Theta(e log e)
- Assuming connected graph
  - v - 1<= e <= v^2
  - log v <= log e <= 2 log v
  - log e = Theta(log v)
- Total runtime = Theta(e log e) = Theta(e log v)
- Same runtime as Prim's

Problem of the Day: Weighted Shortest Paths



1.6M vertices, 3.8M arcs, travel time metric.

Source: https://www.cs.princeton.edu/courses/archive/spr09/cos423/Lectures/reach-mit.pdf

Problem of the Day: Weighted Shortest Paths

- **Input**: starting and destination addresses and a road network
  - Road segments and intersections
  - Road segments are labeled by **travel time**
- **Output**:
  - A **shortest path** from starting address to destination address

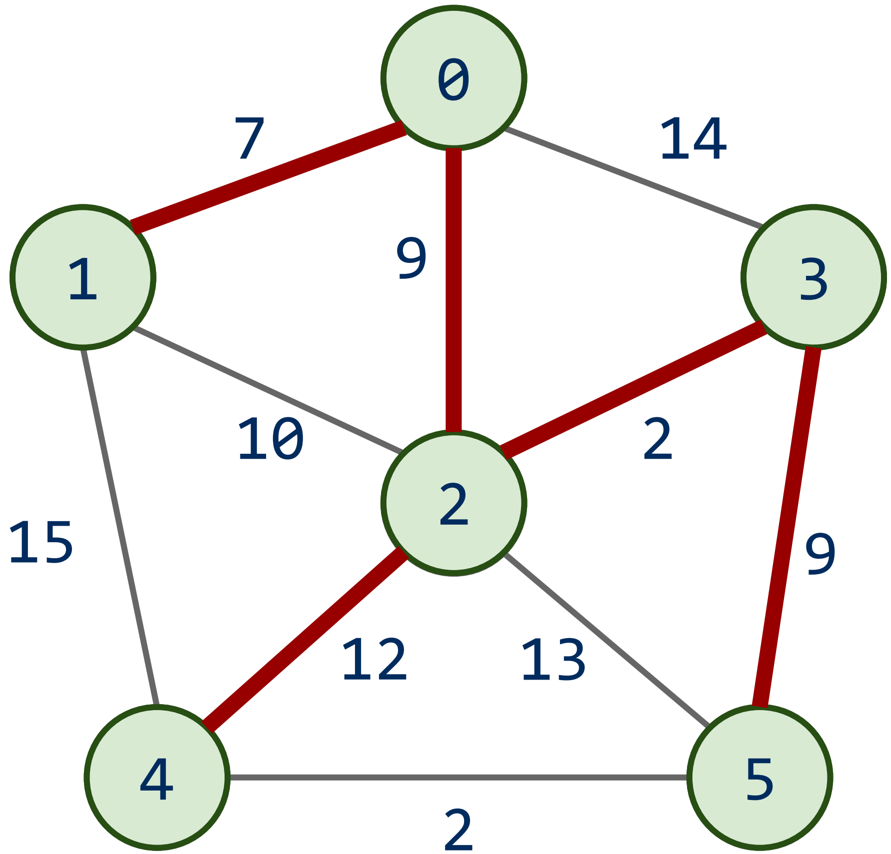
Dijkstra’s algorithm: Data Structures and Initialization

- distance[]: **best known** shortest distance from start to each vertex
- distance[start] = 0
- distance[x] = Double.POSITIVE_INFINITY for other vertices

Dijkstra’s algorithm: Data Structures and Initialization

- cur = start
- While destination not visited:
  - For each **unvisited** neighbor x of cur
    - Compute **shortest** distance from start to x **through cur**
      - = distance[cur] + weight of edge from cur to x
    - Update distance[x] if distance through cur < distance[x]
  - Mark cur as visited
  - cur = an unvisited vertex with the smallest distance

Dijkstra's example



	Distance	Parent
0	0	--
1	7	0
2	9	0
3	11	2
4	21	2
5	20	3

Notes

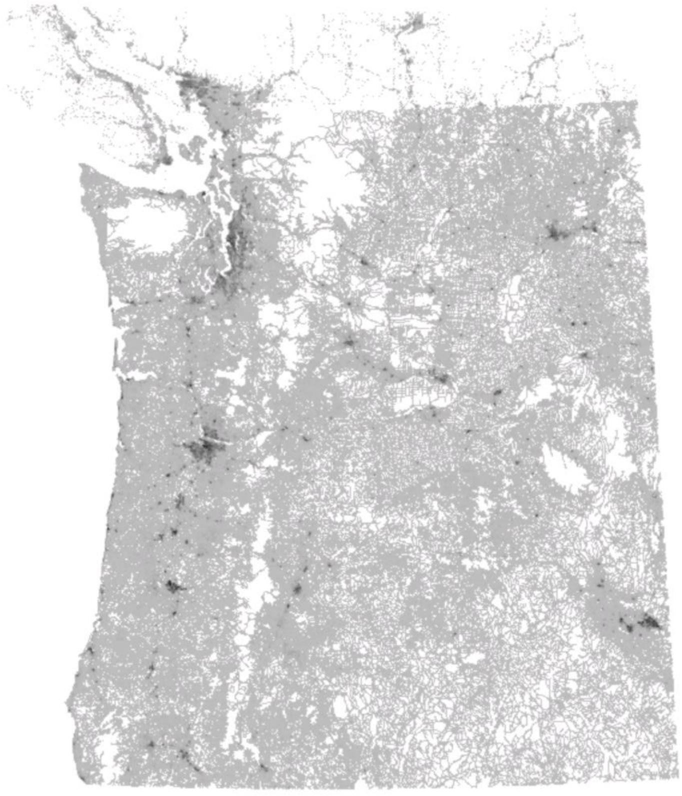
- The distance array keeps track of the **best path** from start
  - Compare to **best edge** array in Prim's
- Once a vertex is **visited**, its distance value **doesn't change**
- **Parent** array used to construct a shortest path

Analysis of Dijkstra's algorithm

- Depends on implementation!
  - Distance and parent arrays? **Theta(v^2)**
  - PQ?
    - very similar to **Eager Prim's** **Theta(e log v)**
      - Storing best paths instead of best edges
- This is worst-case runtime
- Algorithm may **stop earlier** when destination visited
- Order of selecting vertices matters!

Dijkstra's Real-World Optimizations

- **Real-world road networks:**
  - millions of vertices and edges
  - want fast response (<100 ms)

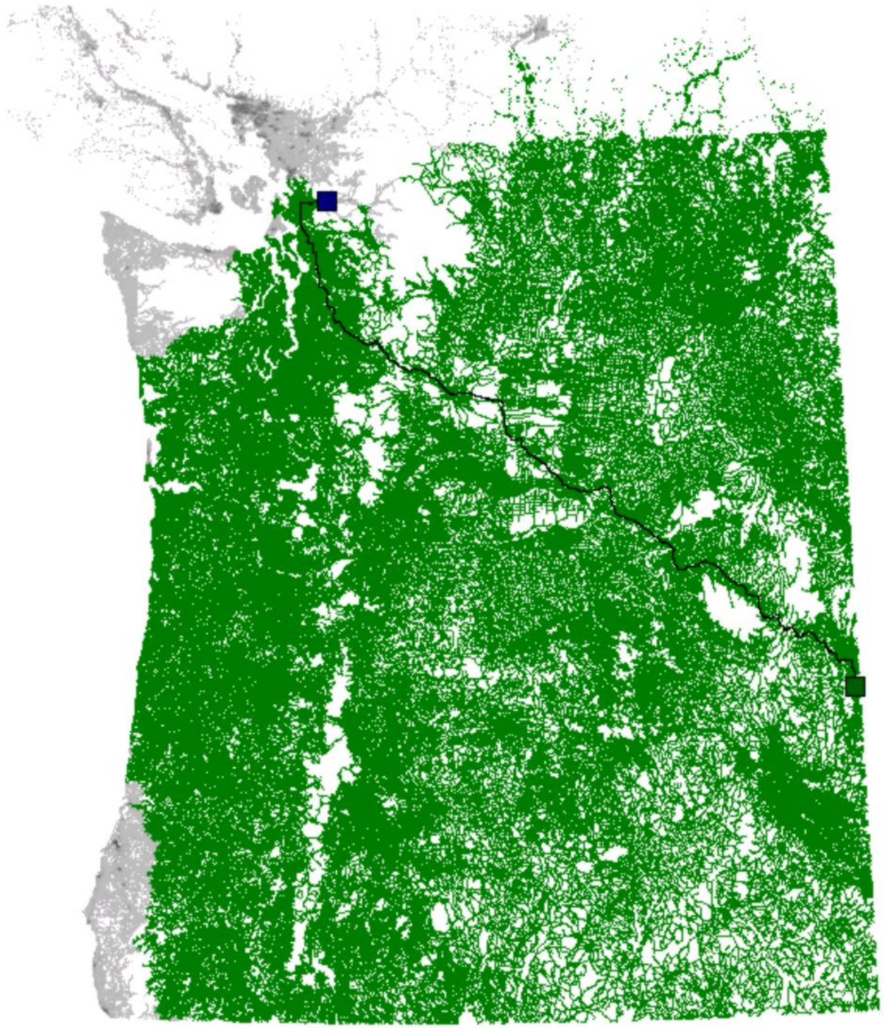


1.6M vertices, 3.8M arcs, travel time metric.

Source: https://www.cs.princeton.edu/courses/archive/spr09/cos423/Lectures/reach-mit.pdf

Dijkstra’s is too slow for such big graphs

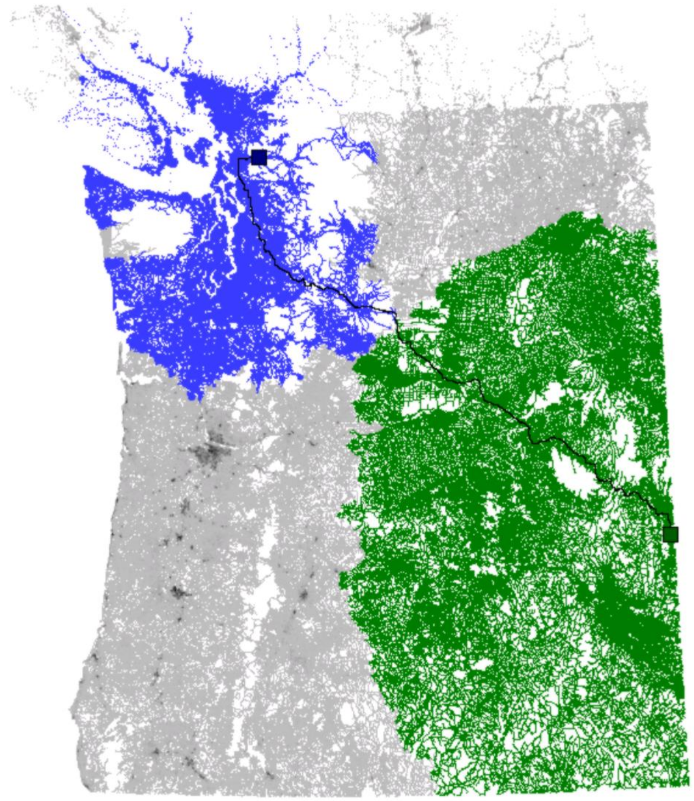
- Dijkstra’s will visit so many **not needed** vertices



Searched area

Optimization 1: Bidirectional Search

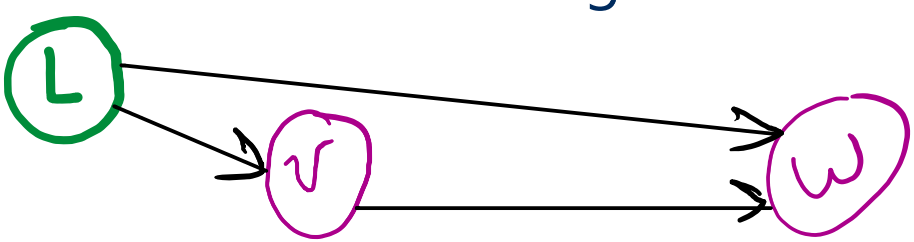
- start two instances of Dijkstra's possibly **in parallel**
  - from source on **original** graph
  - from destination, on **reverse** graph
- When processing an edge to a vertex visited by the other instance, update **shortest known** distance between start and destination
- **Stop** when tops of both heaps give a distance >= shortest known



forward search/ reverse search

Optimization 2: A* Search

- Use **lower-bound estimates** for the distance of the **rest** of the path to destination
- Modified Dijkstra's
  - Pick vertex with minimum **distance[v] + estimate[v]**
- Lower-bound estimates using **landmarks** and **triangular inequality**



$$LW \leq LV + VW$$

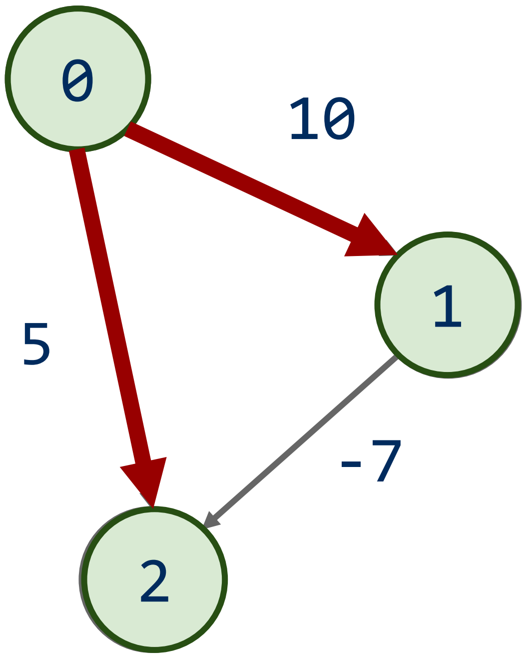
$$VW \geq LW - LV$$

- Requires **preprocessing** to compute and store distances from each vertex to each landmark

Negative Edge Weights

- Modeling of some problems results in graphs with **negative** edge weights
- Example: Planning with **uncertainty**
  - Find a path with **highest probability** of reaching a goal from a starting state
  - **vertices**: milestones; **edges**: actions; **edge weights**: probabilities
  - Find a path with **highest product** of edge weights
  - How to model that as a shortest path problem?
    - log of product is sum of logs
    - maximize x means minimize -x
    - replace each edge weight p by (-1 * log p)
    - find a shortest path in the resulting graph!

Dijkstra's example with negative edge weights



	Distance	Parent
0	0	--
1	10	0
2	5	0

Incorrect!

Dijkstra’s algorithm is incorrect with negative edge weights

Dijkstra’s is correct only when all edge weights $>= 0$

Bellman-Ford’s algorithm: Data Structures and Initialization

- distance[v] = Double.POSITIVE_INFINITY
  - for all vertices except start
- distance[start] = 0

Bellman-Ford's algorithm

- Repeat v-1 times
  - For each vertex cur:
    - For each neighbor x of cur:
      - Compute shortest distance from start to x via cur
        - = distance[cur] + weight of (cur, x)
      - if computed distance < distance[x]
        - Update distance[x] and parent[x]

Runtime of Bellman-Ford's

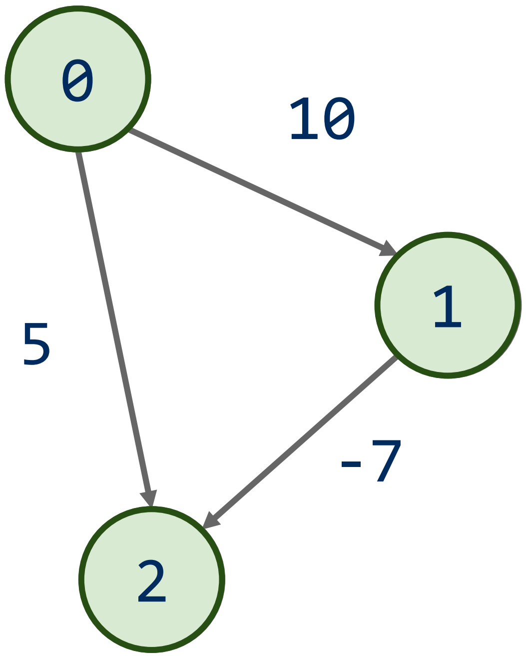
- Repeat v-1 times
  - For each vertex cur:
    - For each neighbor x of cur:
      - Compute shortest distance from start to x via cur
        - = distance[cur] + weight of (cur, x)
      - if computed distance < distance[x]
        - Update distance[x] and parent[x]

- Runtime?
  - O(v*e)

Bellman-Ford’s algorithm: an optimization

- Initialize a FIFO Q; add start to Q
- While Q is not empty:
    - cur = pop a vertex from Q
    - For each neighbor x of cur:
        - Compute shortest distance from start to x via cur
            - = distance[cur] + weight of (cur, x)
        - if computed distance < distance[x]
            - Update distance[x] and parent[x]
            - add x to Q if not already there

Bellman-Ford's example with negative edge weights



	Distance	Parent
0	0	--
1	10	0
2	3	1

FIFO Q:

0
1
2

Correct!

Analysis of Bellman-Ford's algorithm

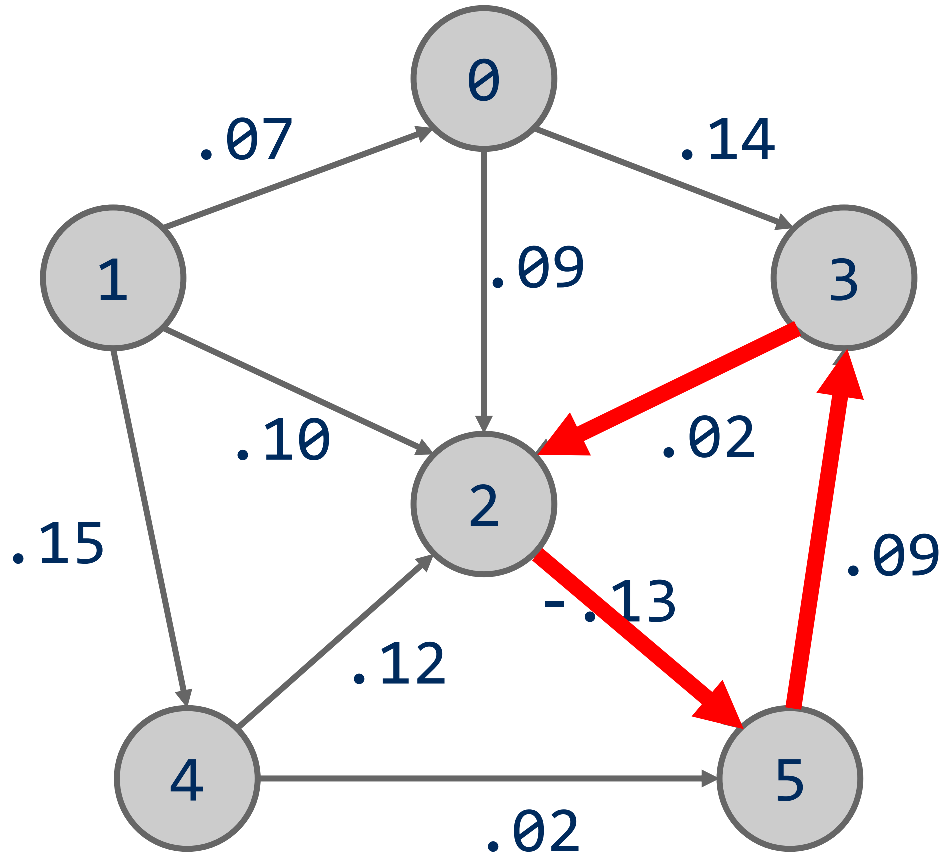
- Bellman-Ford's is correct even when there are negative edge weights in the graph but what about negative cycles?
  - a **negative cycle** is a cycle with a negative total weight

Negative cycles

- **Detecting** such cycle is needed:
  - Bellman-Ford **won't terminate** if a negative cycle exists
  - Some problems can be solved by detecting a negative cycle
- Example: Finding **arbitrage** in currency trade
  - **vertices**: currencies; **edge weights**: exchange rates;
  - **goal**: find a cycle with a **product** of exchange rates that is > 1
    - log of product is sum of logs
    - maximize x means minimize -x
    - Replace each edge weight r by (-1 * log r)
- a cycle with a product > 1 → a **negative cycle** in the resulting graph

Bellman-Ford's example with a negative cycle

- Can you find a negative cycle?



Find a negative cycle reachable from start

- Repeat v-1 times
  - For each vertex cur:
    - For each neighbor x of cur:
      - Compute shortest distance from start to x via cur
        - = distance[cur] + weight of (cur, x)
      - if computed distance < distance[x]
        - Update distance[x] and parent[x]
- If **another iteration** results in update of distance[v] for a vertex v, then v is in a negative cycle
  - can find the cycle using parent values starting from parent[v]

Find a negative cycle reachable from start

- May be able to detect a negative cycle **earlier**
- Build a graph using **parent to child links** set by Bellman-Ford's
- Modify **DFS** to detect if a cycle exists
    - if a **neighbor** already **visited** and is **on** the runtime **stack**
        - we have a **cycle**
        - follow parent links until back to current node
        - add up edge weights
        - if negative stop; otherwise continue